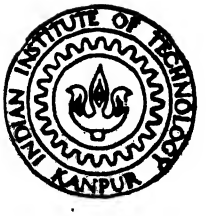
AN IMPLEMENTATION OF CONCEPTUAL GRAPHS

by

N. L. JAGADISH BABU



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

JANUARY, 1988

AN IMPLEMENTATION OF CONCEPTUAL GRAPHS

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

N. L. JAGADISH BABU

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

JANUARY, 1988

CSE-1988-M-BAB-IMP

CERTIFICATE

This is to certify that the thesis entitled **An Implementation of Conceptual Graphs** is a report of work carried out under my supervision by N. L. Jagadish Babu and that has not been submitted elsewhere for a degree.

Rajeev Sangal 27/1/88

Dr.R. Sangal
Asst. Professor
Dept. of C.S.E.
I.I.T.,Kanpur.

Place: Kanpur

Date : January 1988.

ACKNOWLEDGEMENTS

I am greatly indebted to Dr. Rajeev Sangal for his guidance and cooperation throughout my thesis. I am very much grateful to Dr. Vineet Chaitanya for his constant help and guidance.

I thank all my friends who made my stay at IITK enjoyable. I particularly acknowledge the group "H-TOP". I also thank PDP and Bhatta for their moral support to me during my stay at IIT. I also thank P. A. Kishore, who made me think about many things, for the arguments we had.

(N. L. Jagadish Babu)

ABSTRACT

This thesis is about implementation of conceptual graphs. The implementation covers canonical derivation and plausible truth derivation using conceptual graphs. Mechanisms for expanding and contracting types is also implemented. An application of canonical derivation (resolving ambiguities in natural language processing) is also implemented.

CONTENTS

Chapter 1

INTRODUCTION

1.1 Motivation

The motivation of this thesis is to develop a knowledge representation scheme for natural language processing. A knowledge representation scheme for natural language processing is to be used for the following purposes.

(1) to add the background knowledge. Most of the time, a natural language sentence assumes a lot of implicit knowledge. Seeing the sentence, a lot of background has to be used to be able to interpret it.

(2) To remove ambiguities. A natural language sentence may mean different things depending on the context and the words it has. After alternative parses are obtained, the background knowledge can be used to select most appropriate parse.

(3) to reason regarding actions and states. Each natural language sentence will have implications which may make some state changes of the world that is being represented.

1.2 Problems of Knowledge representations

A Knowledge representation scheme should possess the following properties.

(1) Representational adequacy: It is the ability to represent everything of the domain that is to be represented.

(2) Expressive Power: The ease with which a given knowledge is expressed in the scheme by a user is also important.

(3) Inferential ability : The ability to derive new information using the existing information is very important.

(4) Inferential efficiency : The inferences should be made with reasonable efficiency.

(5) Acquisitional efficiency : The ability to acquire and add new knowledge into the system. Ideally, the system should be able to control the flow of new knowledge into the system.

1.3 Our solution

The knowledge representation we have selected for implementation is called conceptual graphs. In them, we have implemented canonical derivation and schema application of conceptual graphs. The canonical derivation implements selectional restrictions and can be used to resolve ambiguities during natural language processing. The ability to abstract (that is in the form of type definition) is also

1.4 Outline of the thesis

Chapter 2 describes conceptual graphs and compares it with other knowledge representation schemes.

Chapter 3 describes in-memory representation of the data structures used for the implementation. This chapter describes conceptual graphs and type hierarchy in memory.

Chapter 4 describes the implementation of canonical derivation. Chapter 5 describes the way to define new concepts using primitive concepts. It also describes the ways these definitions can be used to expand and contract graphs. Chapter 6 describes the input, output parsers for conceptual graphs.

Chapter 7 suggests an application of canonical derivation.

The last chapter, i.e. chapter 8, is about conclusions of the thesis

Chapter2

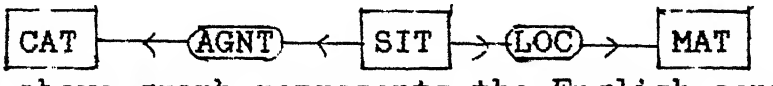
CONCEPTUAL GRAPHS

Description of conceptual graphs

Conceptual graphs is a knowledge representation scheme. This scheme represents knowledge in the form of conceptual graphs and a type hierarchy. It has a set of rules to interpret the knowledge known as formation rules.

Conceptual graphs are bipartite graphs. The two different kinds of nodes in conceptual graphs are concepts and relations. In conceptual graphs, concept nodes represent entities, attributes, states and events and relations show how these concepts are interconnected.

A sample graph is shown below.



The above graph represents the English sentence 'a cat is sitting on a mat '. Square boxes are concepts and circles are relations. CAT, SIT and MAT are the concepts and AGNT and LOC connect them. The relation agnt says that the agent of sit is cat. Similarly for loc, the location of sit is mat.

Conceptual graphs can be defined more formally as follows.

Conceptual graph is a finite, bipartite graph.

the two kinds of nodes are concepts and relations.

Every relation has one or more arcs. Each of these arcs is connected to a concept.

A single concept can be a graph. But every arc of a relation should be connected to a concept.

2.1.1 Description of a concept

A concept can be a physical entity like CAT, TOMATO which represents a typical cat or tomato or it can be an abstract entity like PRICE and JUSTICE. Every concept in a conceptual graph has a type label. In the sample graph the type labels are CAT, SIT and MAT.

2.1.1.1 Type hierarchy

The sample conceptual graph shown says that a cat is sitting on a mat. But this by itself does not say anything about a cat being an animal and sitting is an action and so on. These facts are represented in a type hierarchy in conceptual graphs.

The type hierarchy for conceptual graphs is as follows.

The type hierarchy is a partial ordering defined over the set of type labels of concepts. The symbol <= defines the ordering.

Let s,t be two type labels, then the following is true.

If s <= t then s is known as subtype of t and t is supertype of s.

The set of all possible instances of a type is known as denotation of a type. If δs is denotation of type s, δt is denotation of t and s <= t then $\delta s \subseteq \delta t$.

For example Cat is subtype of Animal. This means that all cats are animals.

2.1.1.2 Individuals

In conceptual graphs, type label says only a part of what a concept is. The part is called referent of a concept. The name referent comes from the fact that it tells exactly which member of the type the graph is referring to. The referent can be one of the following.

A generic referent, written as *, which represents a typical member of the type.

An individual referent, which represents a particular member of a type. This is represented by the name of individual, in the concept it is followed after the type label with a colon.

A set referent which represents a group of the type of the concept.

2.1.2 Description of Relation

A relation determines the interconnection between concepts in a graph. It is distinguished by its type label. A relation with a type label will always have the same number of arcs to concepts.

Two relations r and s are said to be duplicate if and

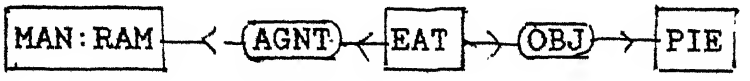
only if

type labels r and s are equal.

if i-th arc r is connected to concept a, then i-th arc of s should be connected to concept a for all i.

2.1.3 Mapping of conceptual graphs to logic

A conceptual graph can be converted into standard logic as follows. For each concept let there be a predicate with its referent as its argument. If a concept's type is man and its referent is Ram, then man(ram) would be the logical equivalent. For each relation, let there be a predicate with relation type as predicate name and referents of the concepts are arguments. All generic referents are existentially quantified. All such predicates are and-ed to get the logical equivalent. An example is shown below.



can be written as follows

$\exists x \exists y (MAN(RAM) \wedge AGNT(RAM,X) \wedge EAT(X) \wedge OBJ\ (X,Y) \wedge PIE(Y))$

2.1.4 Formation rules

Formation rules are rules which create a new set of graphs from a given set of graphs. They are as follows.

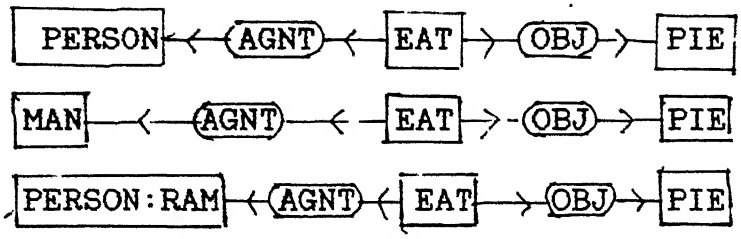
2.1.4.1 Copy

Every graph is a copy of itself.

2.1.4.2 Restrict

The restrict rule represents the set of applicable

instances for a concept in a graph. This is done either by replacing a type label of a concept with its subtype or by replacing a generic referent by an individual referent.

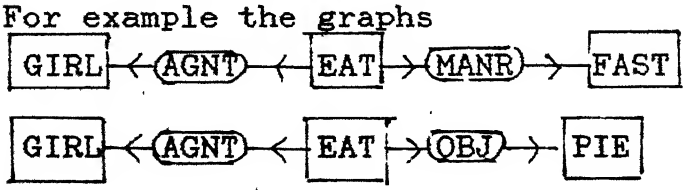


In the above example ,figures 2 and 3 are restrictions of figure 1.

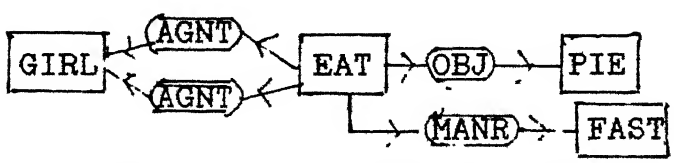
2.1.4.3 Join

Join operation makes two concepts in two different graphs as identical. When two concepts are merged, all of the arcs of the concepts merged are to be connected to the resultant of the merger. The join can also be applied to the concepts of a single graph. The type labels of the two types to be merged is replaced by the least common subtype of the two types.

For example the graphs



can be joined to



The concepts GIRL and EAT in a graph are merged with the respectie concepts in the other graph.

2.1.4.4 Simplify

With this operation the duplicate relations are removed. For example,

can be reduced to

2.2 Description of canonical derivation

2.2.1 Notation of generalization and specialization

Given a graph and a graph that is obtained by restrict operation. If t1 is the type label which is restricted to t2 , then we know that

t2 (x) -> t1 (x) (for any x)

This implies that if the graph g2 derived by restricting a type t1 in a graph g1 then if g2 is true then g1 is true.

Similarly for join ,

A∧B -> B

A∧B -> A

Using the above rules, we know that if a graph is true, then the graphs from which it is derived are also true.

The graphs obtained by applying the formation rules are

known as the specializations of the graphs from which they are derived. If g1 is specialization of g2 then g2 is known as generalization of g1. Further, if g1 is true then g2 is also true. But, if g2 is true then g1 may not be true.

Thus we see that formation rules are not truth-preserving. But, if a graph is meaningful or valid , then the graph derived from it are also valid. Thus the formation rules can be used to derive new graphs from a set of valid (canonical) graphs. If a set of graphs known as canonical basis is available from which all meaningful graphs can be derived, then all graphs can be validated using this forma-tion rules. The formation rules are thus used to enforce selectional constraints. The formation rules are also referred to as canonical operators in this thesis.

2.2.2 Inference in conceptual graphs

In conceptual graphs there are three levels of infer-ence. The first is canonical derivation as described in last section. This enforces selectional constraints for conceptual graphs. The second level is plausible truth derivation. Here a set of properties of typical members of each concept (these are known as schemas in conceptual graphs) are used to determine possibility of a graph. The next level of inference is logical inference which says whether a given graph is true or not.

2.3 Comparison with other knowledge representation schemes

This section compares conceptual graphs with other

schemes in the criteria mentioned earlier.

Two of these criteria are representational adequacy and expressive power. Conceptual graphs are adequate to represent and for most of the natural language the conversion to conceptual graphs is direct. Compared to this other knowledge representations are not adequate. Semantic networks do not distinguish between a typical member of a type , a type and a set nodes. Frames do not distinguish between definitional conditions of a type and possible conditions of a type . Conceptual graphs distinguish these by having provisions for type definitions and schemas.

Inferential ability is other criteria for comparison. Only logic is theoretically sound and complete. Conceptual graphs have direct mapping to first order logic. Things normally represented by model logic involving time and context are handled naturally by conceptual graphs. Further, conceptual graphs have a powerful inference mechanism as described in the last section.

Acquisitional efficiency is controlling the knowledge entering the system. In conceptual graphs, the knowledge entering the system can be controlled. The new schemas can be validated by canonical derivation. New facts can be verified by plausible truth derivation. Other representation schemes do not have this kind of inference mechanism which enables such a control.

Chapter 3

REPRESENTATION IN MEMORY

This chapter describes the representation of conceptual graphs in memory. While designing this representation the main consideration was to make algorithms faster rather than smaller memory space. This is done taking in view the large amount of computations that are to be done in a typical knowledge base system. However the total memory requirements have been kept to affordable limits. A sample memory requirements are projected.

The chapter is divided into two parts. The first part is about the representation of conceptual graphs. The second is about representation of type-hierarchy.

3.1 Representation of graph :

While designing a representation for a graph, it is assumed that the following operations are performed frequently.

1.Return all the relations in a graph

2.Return all the concepts connected to a relation in a graph

3.Return all the relations connected to a concept in

a graph

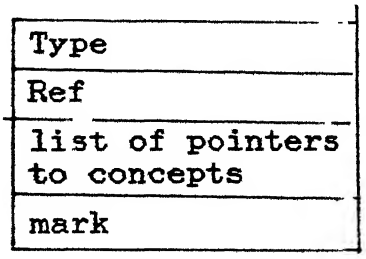
4.whether a relation set is subset of another set.

The actual application of the above operations would be shown in the next chapter.

The basic representation of the graph has been to have records as nodes and pointers as arcs in the graph. The pointers are stored in the records representing the nodes. The individual records for the components of the graph i.e. concepts and relations is described in the next sections.

3.1.1 Concepts:

A concept, shown pictorially below, is represented as a record with the following fields. The field type is the type of the concept. The field Ref is the referent of the concept. The next field is a pointer to the list of relations it is connected to. The last field called mark is reserved for the marking of a concept. Marking is used in canonical derivation.



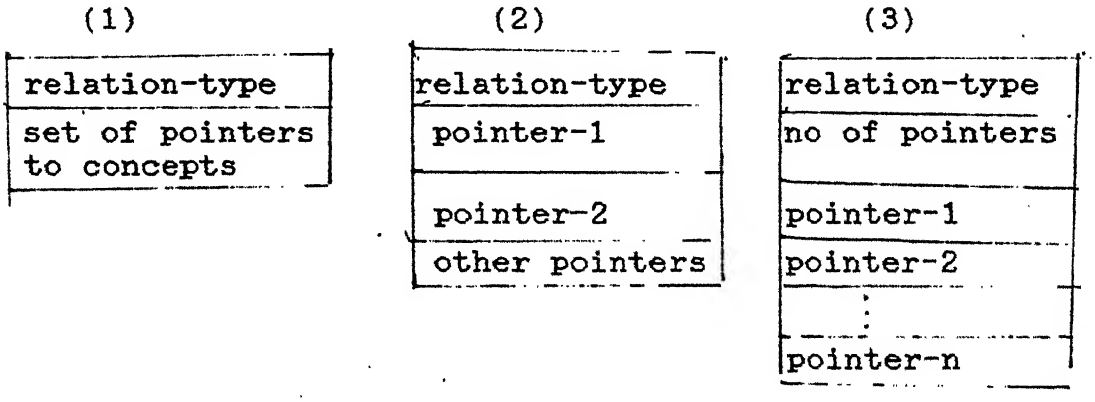
The last of these fields is for marking the concepts which is used by the graph comparing algorithm.

3.1.2 Relations :

Keeping in view the representation of concept, all that

necessary is the relation type to represent a relation. However, it might often be necessary to get the list of all concepts attached to a relation without actually traversing the whole graph. Similarly it is also useful to be able to respond with all the relations connected to a concept without traversing the graph. To have an efficient solution, both the pointers are necessary. As a result of this, an additional validation check is necessary to make the graph consistent.

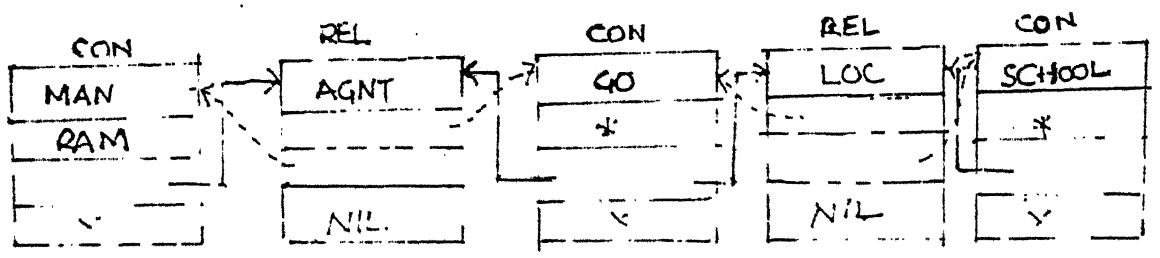
The following representations were considered



In all the above representations for the relation a pointer points to a concept. In representation 1, a pointer to a list of n pointers to concepts is stored in the record for the relation. When there are n concepts connected to a relation the pointers numbered from 3 to n are represented in a list in representation 2. All the n pointers are represented in a list in representation 1. In the third representation each has a distinct field and there is an additional field which gives the number of concepts to that relation.

The first of these is the simplest but also the most inefficient in that to access any pointer it has to traverse all the pointers before it. The last of these of variant record type and gives a direct access to all the pointers. Since it would be difficult to handle variant-length records, a middle representation is found. This representation favours binary relations to n-ary relations. Since binary relations are more common than other n-ary relations this is preferred. This is reasonably efficient and is simpler than the variant-length one.

A typical example to illustrate this representation is given below.



Some of the existing implementations used tags with each record[Sowa 86]. In the above representation , there are no tags. This is not necessary as the graph is bipartite. A relation cannot point to another relation and a concept cannot point to another concept. The validation of the graph can be done even without these tags by checking these relations.

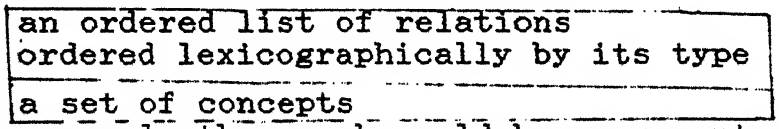
3.1.3 Graph :

Having represented concepts and relations, the simplest

way to represent a graph is to have a head pointing either to a relation or to a concept. However it might lead to more efficient representation if we consider the following.

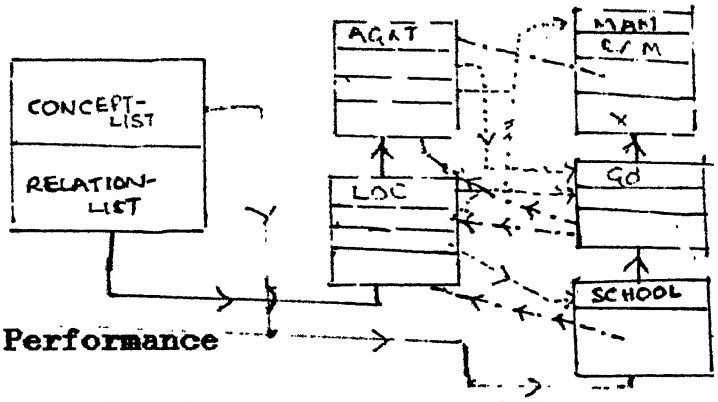
For instance, it is often necessary to find all the concepts in a graph and all relations in a graph without traversing the graph. Further the relations-set can be ordered using its type name to make efficient use. This would make comparison of two graphs as illustrated in the following chapters. For concepts it is not necessary to order because its hierarchy is important.

So the representation that has been chosen for a graph is given below.



For example the graph would be represented as

Ram went to School.
[man:ram] <- (agnt) <- [go] -> (loc) -> [school].



3.1.4 Performance

This sub-section presents performance analysis for the various operations listed in the beginning of the section. Let a graph have r relations and c concepts.

The first operation, i.e. all relations a graph has,

can be performed in O(1) time. This is because the structure for the graph has a pointer to the list of relations.

The next operation is to find all concepts connected to a relation. This can also be done in constant time as every relation record has pointers to the concepts it is connected to. Similarly for the next operation, i.e. the relations connected to a concept, the time is constant.

The last operation is to test whether a relation-set of a graph is a subset of relation-set of another graph. This operation is needed while comparing two graphs. This can be performed in O(r1 + r2) time because the relations are ordered by their type name, where r1 and r2 are the number of relations the relation-sets have. The same operation without the ordering of relations can be executed in O(r1 * r2), where r1 and r2 are relation sets.

This reduction to linear time is possible because of the ordering on the relation-sets. This is at the cost of the performance of the operation of adding a new relation to the graph. Whenever a relation is added, it has to be placed in lexicographic order of the relation-set. However, the frequency of this operation is very low compared to the subset operation, the described implementation is chosen.

3.2 Type hierarchy:

The following operations are important for a type hierarchy.

1. whether a given type is subtype(supertype) of another

type

2. All subtypes of a given type

3.All supertypes of a given type.

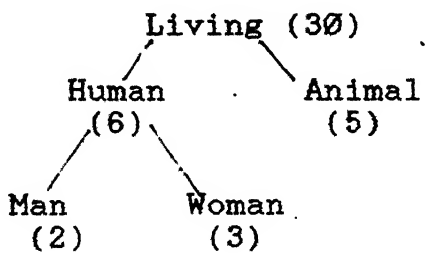
3.2.1 Basic Representation :

For every type there is a record containing pointers to immediate subtypes and immediate super types. Both the pointers are essential to ensure efficient executions of operations 2 and 3.

By having pointers to immediate subtypes and immediate supertypes alone, the first operation is not efficient. To check whether a given type is subtype of another given type, we have to go through the immediate subtype supertype pointers. In either case there is a search involved. To avoid the search the idea of factorization and family names is introduced.

3.2.2 Factorization

The factorization idea is to give each type an integer number. The relation divisibility would be used to represent subtype. For example consider the hierarchy



The numbers in brackets are the numbers assigned to the type. If the thing to be checked is whether man is subtype

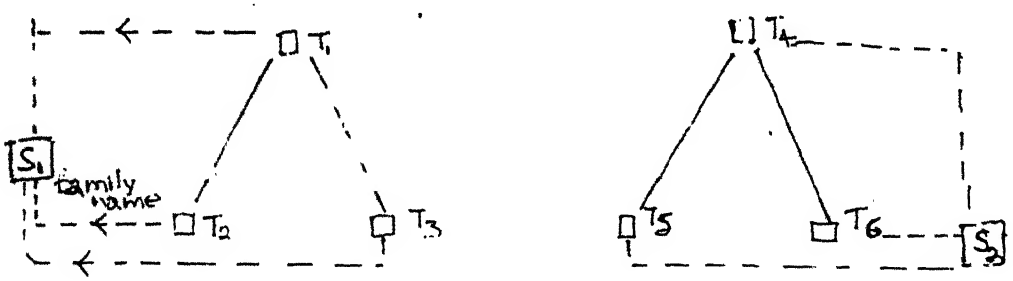
of living , it is checked by dividing the number assigned to type living by the number of type man. This idea can be extended to any hierarchy if a prime number is assigned to a leaf. However this could not be used due to the limited integer precision of present day computers. Because the numbers are multiplied to get number assigned to a super-type, the number would quickly increase to maximum permis-sale integer if the type-hierarchy has more than a small number of levels. Therefore, we use the idea of family names.

3.2.3 Family names

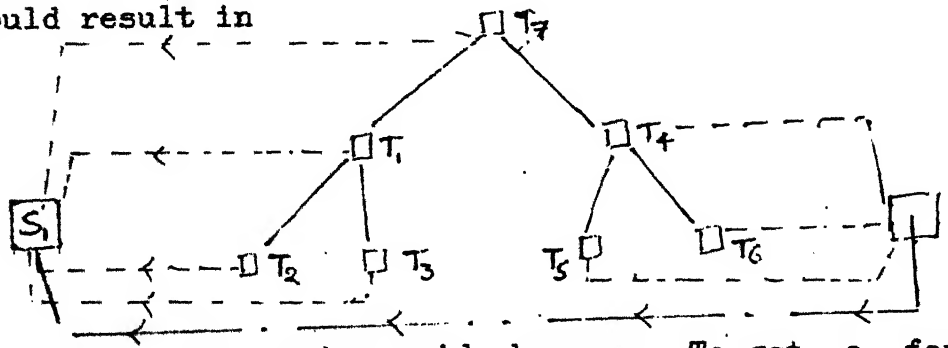
Family names allow us to avoid a search when two given types are not connected at all (that is, they belong to two different trees). Each type (not including the universal type T and the absurd type) occurring in a tree is given a family name that is stored in the record for the type. If the family names are different, we are assured that they are not connected by subtype supertype relation. If the family names match, then a search is conducted to find whether they are related in a subtype relation. This mechanism is useful when there are large number of disconnected trees in the type hierarchy (not counting universal type T and absurd type), an observation found to be true in practice.

However this mechanism has a problem when the hierarchy is not static. In the course of usage of knowledge base,

some new type may arise which may connect hitherto two different trees. The family names are given to new types as follows. If a new type is added, it can be given a family-name for the tree to which it is connected or a new family name if it is connected to none. If a new type is added such that it connects two different trees, the two trees should have the same family name. The family names of all types belonging to one tree have to be changed. To avoid this, all types in a family are made to share a memory structure that will have either a family name or a pointer to another similar memory structure. This mechanism is necessary to avoid searching the tree whenever a new type is added to change the family name of each type it is having. This can be illustrated by the following example.



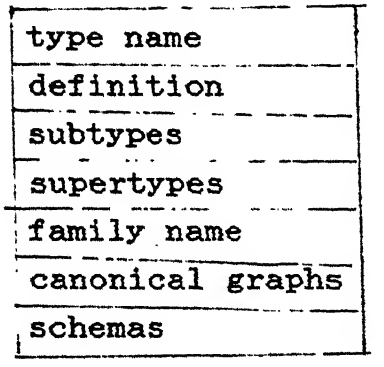
If the above two hierarchies are to be connected , they would result in



There is a price paid, however. To get a family name now, it would require two accesses. As the hierarchy builds

the indirect pointers may become a overhead. So a periodical purge is necessary to remove these indirect pointers.

The final representation for concept-type record, therefore is as follows



The definition field and the last two fields are explained in the next sections.

This completes the in memory representation of conceptual graphs.

Chapter 4

CANONICAL DERIVATION

The canonical graphs are graphs that are used for enforcing selectional constraints. Given a set of canonical graphs or canonical basis, it can be verified whether a given graph is canonical or not. This is done by deriving the graph from canonical basis by using the four canonical operators. To apply the four canonical operators on a given canonical basis and derive a graph would be difficult as the number of graphs that are to be considered are very large. However, this can be simplified by the fact that, it is not necessary to know the sequence of canonical operations to be performed to obtain the derivation. It is sufficient to know that a graph can be derived. However, the derivation described below can give the set of graphs that make the derivation. Before describing the derivation following observations ought to be noted.

Each graph in canonical basis is a generalization of the given graph. This is because, application of any of the four canonical operators would result in a specialization of a graph.

If g1 >= g, g2 >= g (where >= stands for

The algorithm implemented tries to check whether a projection operator can be applied between two graphs. It checks this by checking all concepts and all relations follow the above properties. The algorithm compares the two graphs relation by relation. It checks if the two relation types are equal. If they are, it checks concepts connected to the two relations match. If two relations are connected to the same concept in the generalization they should be connected to the corresponding concept in specialization. To check this the mark field in the concept record is used. The same mark in two concepts of the two graphs would mean that they are corresponding concepts. The complete algorithm is as follows.

4.1.1 Algorithm

```
algorithm match-graph (g1 g2)
step 1) For each relation r1 in g1  do
   1.1 find corresponding relation r2 in g2 with same type
   1.2 do for each such relation r2
   1.3 for each concept c1 connected to relation r1 in g1
     1.3.1 Check if corresponding concept
      c2 connected to r2 in g2 is subtype of the concept c1
           else fail and  try next r2.
     1.3.2 if the marks field is already marked in c1
        1.3.2.1 check for the same mark in the corresponding c2
          if marks field is not same then try next r2.
              else mark both the concepts of g1
              and g2 with a new marker.
end match-graph.
```

4.1.2 Analysis

The time analysis of this algorithm is as follows. Let

generalization),then there exists a graph G which is a subgraph of g and which is derivable from g1 and g2.

If g1 >= g, g2 >= g, ... gn >= g and R1,R2 ... Rn are relation sets of g1, g2, ... gn respectively and R relation set of g, then if R1∪R2∪ .... Rn = R , Then g can be derived from g1, g2 ... gn alone.

The canonical derivation of a graph is equal to finding the generalizations of the graph, which are complete to make a derivation. So the basic operation that is to be provided is checking whether a given graph is specialization of another given graph.

4.1 Graph comparing algorithm

The implementation of graph-comparing algorithm is based on the following theorem [Sowa 84].

For any conceptual graphs u and v where u <= v, there must exist a mapping π: u -> v , where πv is a subgraph of u called projection of v on u. The projection operator has the following properties.

For any concept c in v, πc is a concept in πv where type (πc) <= type (c) If c is a individual concept then referent (πc) = referent (c).

For each relation r in v, r is a relation in πv where type (πr) = type (r) If ith arc of r is connected to a concept c in v, the ith arc of πr is connected to πc in πv.

jection operator can be applied between two graphs. It checks this by checking all concepts and all relations follow the above properties. The algorithm compares the two graphs relation by relation. It checks if the two relation types are equal. If they are, it checks concepts connected to the two relations match. If two relations are connected to the same concept in the generalization they should be connected to the corresponding concept in specialization. To check this the mark field in the concept record is used. The same mark in two concepts of the two graphs would mean that they are corresponding concepts. The complete algorithm is as follows.

```
algorithm match-graph (g1 g2)
step 1) For each relation r1 in g1  do
   1.1 find corresponding relation r2 in g2 with same type
   1.2 do for each such relation r2
   1.3 for each concept c1 connected to relation r1 in g1
     1.3.1 Check if corresponding concept
      c2 connected to r2 in g2 is subtype of the concept c1
           else fail and  try next r2.
     1.3.2 if the marks field is already marked in c1
        1.3.2.1 check for the same mark in the corresponding c2
          if marks field is not same then try next r2.
              else mark both the concepts of g1
              and g2 with a new marker.
end match-graph.
```

4.1.2 Analysis

The time analysis of this algorithm is as follows. Let the graph g1 has r relations and c concepts. The worst case

time for the algorithm is 2 * r (h + m).

where h = time taken for subtype check.

m = constant time.

This is because the number concepts compared for a successful comparison of graphs (for two graphs having only diadic relations) is 2 * r.

4.2 The derivation

To get the derivation of the given graph g, the above algorithm is used repetitively. In any canonical basis the number of graphs would be quite large. To avoid comparing with all graphs, the following indexing mechanism is used.

4.2.1 Indexing

The indexing mechanism used is to have an index pointer from each concept type to the canonical graph it occurs. The index is stored in the concept type record. The fields named canonical graphs and schemas are used for this purpose. The field canonical graph stores a pointer to the canonical graph and the field schema stores a pointer to a schema in which the concept type is present.

4.2.2 Algorithm for derivation

The derivation is applied with the graphs indexed by the concepts in the graph to be derived and the graphs indexed by their supertypes. The observations 2 and 3 stated in the beginning of the chapter are used. Each time a

graph is verified as the generalization of the graph to be derived, the corresponding relations of the generalization are marked in the graph to be derived. When all the relations are marked the derivation is complete. The set of graphs which are generalizations can derive the graph. The comparison of graphs to check for generalization is done only if the relation set of the generalization is subset of the that of the graph to be derived.

The algorithm for this is as follows

```
Algorithm Canonical derivation (g)
step 1) Initialize
        R <- relation-set (g)
        U <- R     ;;; U unmarked relations
        M <- {}    ;;; M marked relation set
step 2)
      For each of the concept c in g do steps 3,4 and 5
step 3) I <- index-set (c)
     ;;; index-set is set of all indexed graphs
     ;;; this also include graphs indexed by supertypes of c
step 4) For each graph ig in I do
  4.1 Rig <- relation-set (ig )          ;;ig -- indexed graph
                                  ;;;Rig -- relation set of ig
      if (Rig ⊆ R) and (Rig ∩ U) ≠ {} )
      then
      4.1.1 if ig >= g  ;;; >= is tested by match-graph
       then
         4.1.1.1 U <- U - Rig
                 M <- M   Rig
                 If  (U = {}) then exit-success
end Canonical derivation
```

4.2.3 Analysis

There are many subset operations involving relation sets in the above algorithm. This is the main motivation for relations being ordered in the graph. All these operations can now be performed in linear time to the length of each of

the relation sets. The functions get-relation-set and index-set are direct functions graphs to be compared.

4.3 Schematic Derivations

Schemas are also like canonical graphs. If a graph is a schema, the graphs derived from it are also possible. The same principle of derivation is used. Further a schema is applicable to only one concept. The derivation is possible only by considering schemas of only one concept-type (and its super types). The rest of the algorithm remains the same.

Chapter 5

TYPE DEFINITION, EXPANSION AND CONTRACTION

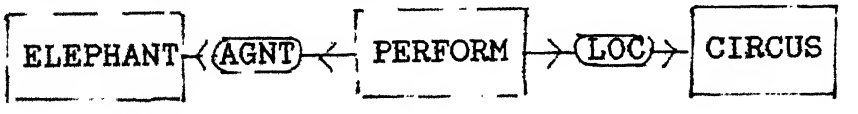
5.1 Description

Type definition allows the user to define new concept types. It provide a way of expanding a concept in primitive concepts or contracting a concept in a graph of primitives. Definitions can be specified for a type in two different ways[Sowa 84] by stating necessary and sufficient conditions for the type, or by giving examples and saying that anything similar to these belong to the type. conceptual graphs support type definitions by stating the necessary and sufficient conditions.

5.1.1 Type definition

A type definition declares that a type label t is defined by abstraction λau , where u is a conceptual graph and is the body of the definition, a is the basis type. or genus of type t. After the type t will be subtype of type a. For example, a type CIRCUS-ELEPHANT is defined as follows.

type CIRCUS-ELEPHANT is



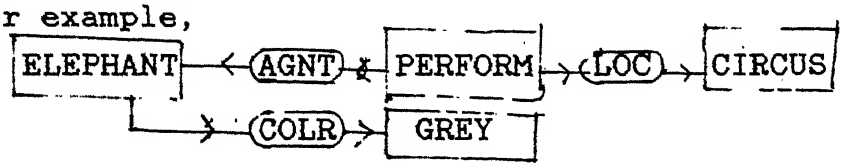
Here the genus type is Elephant, the type circus-

elephant would be subtype of elephant after the definition.

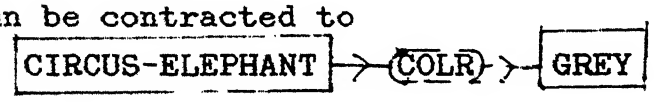
5.1.2 Type contraction

Once a mechanism is available for defining new types, the definitions can be used to simplify graphs. Whenever a graph contains a a subgraph matching the definition, the subgraph can be replaced by the concept of the defined type. This operation is known as type contraction.

For example,



can be contracted to



This would make the graph easier to understand. In type contraction, if some of the concepts are restricted either by having individual referent or by a new type name, that concepts and the subgraph connecting the genus and those concepts is to be retained.

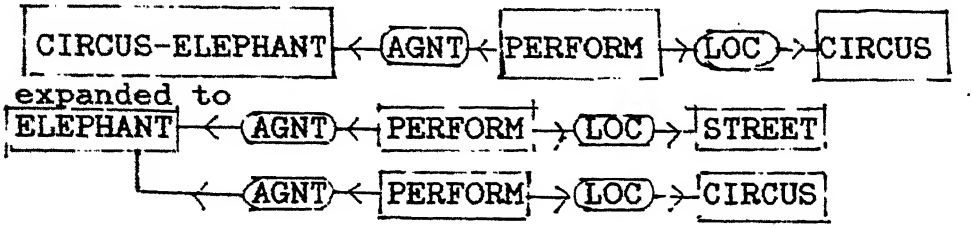
5.1.3 Type expansion

Let u be a conceptual graph containing a concept a where type (a) = \bv then minimal or simple type expansion is joining the two graphs u and v on concepts a and b. The join operation is as described in chapter 2.

This minimal type expansion is a canonical operation. But the result of a type-contraction followed by a type-expansion is not identical to the original graph. A maximal

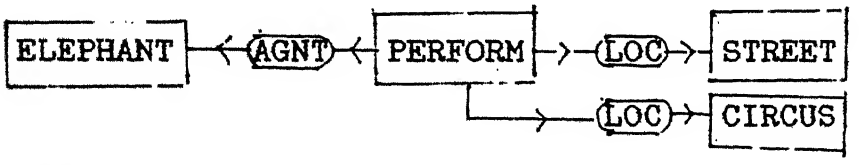
type expansion makes more changes to the graph to restore it back to the original. A maximum type expansion for the graphs as described above is joining graphs u and v on concepts a and b maximally. A maximum join joins all possible concept pairs and removes duplicate relations. The simple type expansion preserves truth. Maximal type expansion may not preserve truth.

To illustrate the difference between the two types of expansion, an example is shown below.



is expanded to

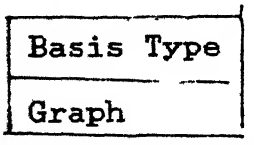
in simple type expansion



in maximum type expansion.

5.2 Implementation

Type definition is stored in the concept-type record. Type definition consists of two parts as illustrated below



The basis type is concept referent pair where the concept is the genus of the type being defined. The graph is definition of the type.

5.2.1 Type expansion

The two forms of type expansion are implemented. The first form is minimal and is implemented by join operation. The join operation adds to one of the graphs, relation by relation of the other graph. Whenever a relation is added, all of its concepts (except the genus) are also added. The maximal join is implemented by merging all relations of same type. The merging is done only if corresponding concepts match.

Algorithms for the above are as follows.

```
Algorithm: simple type-expansion (g, d ,t ,c-g)
                ;;; g is the graph in which type t is to be expanded
                ;;; d is the definition in which genus type is c-g
1. for each relation r-d in d
  1.1 add all concepts (except c-g) connected to r-d to g.
  1.2 add relation r-d to g
  1.3 connect concepts to r-d in g as in r-d in g
  1.4 for the genus concept c-g connect to t
end simple type-expansion

Algorithm for maximum type-expansion (g d t c-g)
          ;; all parameters are as described above
1. for each relation r-d in d
  1.1 check if it matches to any relation r-g in g
    ;;;matching is follows
    ;;;relation types r-d and r-g match
    ;;if i-th concept connected to r-d is of type t-1
    ;; and ith concept connected to r-g is of type t-2
    ;; then t-1 should be supertype of t-2
  1.2 if it matches then do not add this relation to g
  1.3 else add this relation as in simple type expansion
end maximum type-expansion
```

5.2.2 Type contraction

Type-contraction has been implemented as follows. The comparison of the graph with the definition is done by using the graph comparing algorithm of chapter 4. If the result is

success then only the type contraction is applied. All relations corresponding to those in definition are marked to be removed. If any concept connected to any relation is specialized that concept and the subgraph from that concept to genus are unmarked. After this is done for each concept, all marked relations are removed. The genus type name is replaced by the type name with which the contraction is done.

The algorithm for type-contraction is as follows:

```
Algorithm type-contraction (g d t c-g)
     ;; parameters are as described in type expansion
1. Check if d <= g  ;;by graph comparision algo. in chap 3
2. if it is not true then exit-failure
3. mark all relations in g corresponding to some
     relation in d.
4. for each relation r-g
   4.1 for each concept c1-g connected to r-g
   4.2 if corresponding concept in r-d
       is a supertype of it.
       ;;; corresponding concept is the concept
       ;;; connected to relation r-d which is
       ;;; of the same type as r-g
   4.3 if it is, then unmark this relation
       and all relations connected from this concept
       to t.
5. remove all marked relations.
6. rename the type label with the type name
 whose definition is used for contraction.

end type-contraction.
```

Chapter 6

USER INTERFACE

The box and circle notation for conceptual graphs as used in other chapters is not easy to type in. This chapter describes a linear notaaion for graphs which is used for input and output conceptual graphs. The notation is followed by a description of parser and printer for the notation.

6.1 Description of linear notation

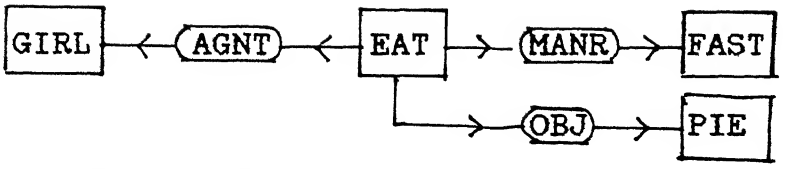
In the linear notation the relations can be monadic or diadic. This is because most of the relations fall in these catogories. In this notation, concepts are represented by square boxes and relations by normal brackets. The arcs are represented by -> and <- symbols.

For example

would be represented as

```
[cat] <- (agnt) <- [sit] -> (loc) -> [mat].
```

If there are more than two relations connected to a concept, the graph is represented as a tree. For example,



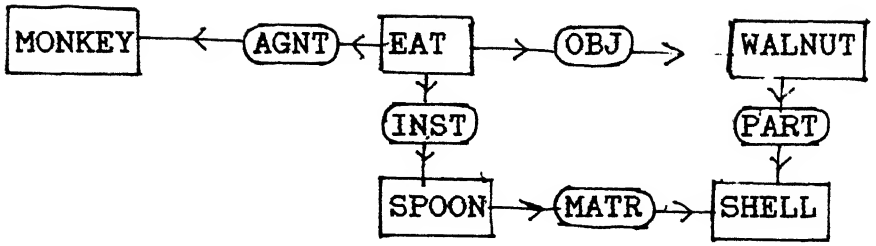
is represented as

```
[eat]-
    (agnt) -> [girl]
    (obj)  -> [pie]
    (manr) -> [fast].
```

The symbol '-' is used together with ',' or '.' is used to represent all relations connected to a concept. The concept that preceeds significance. The same graph can be represented with girl as head :

```
[girl]-
    (agnt) <- [eat] -
                (obj) -> [pie]
                (manr) -> [fast].
```

In the above notation all concepts with same type and referent represent the same physical concept in a graph. If a graph has cycles then to make representations linear, the cycle is broken and the concept which has to occur in two branches is given a pseudo referent *x,*y etc(if it does not have a individual referent). This is illustrated as in the following example.



can be represented as

```
[Eat]-
   (agnt) -> [Monkey]
   (obj) -> [Walnut:*x]
   (inst) -> [Spoon] -> (matr) -> [Shell] ->
                          (part) -> [Walnut:*X].
```

6.2 Implementation of parser and printer

A parser which reads fron the terminal in the linear notation and creates a graph is implemented. It is implemeted using standard parsing techniques. The input string was broken into tokens and the tokens are analyzed for syntactically.

A printer for the above notation is necessary as the graph (described as in chapter 3) has pointers from concepts to relations and vice versa. The printer takes a graph and printers in linar notation. It does the printing by selecting one of the concepts as head (usually the one with more relations) and prints relation by relation. The same technique is applied for every concept connected to a relation.

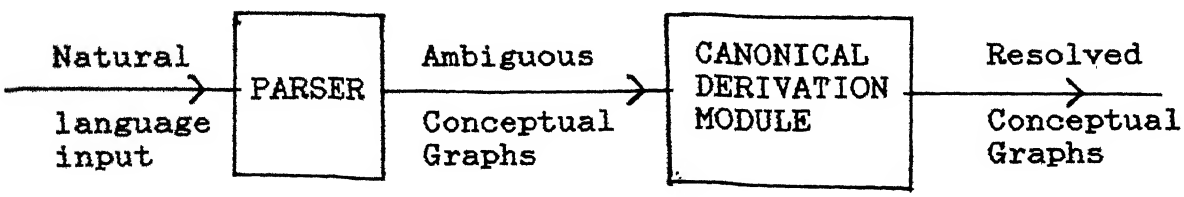
Chapter 7

AN APPLICATION

Conceptual graphs can be used as knowledge representation in AI applications. This chapter describes its use in resolving ambiguities in natural language processing.

7.1 Description

The canonical derivation described in chapter 4 is used for resolving ambiguities in word meanings. The overall scheme is as indicated in the following scheme.



The parser would typically provide a partially constructed graphs to the canonical derivation module which then selects one of the graphs as the right one.

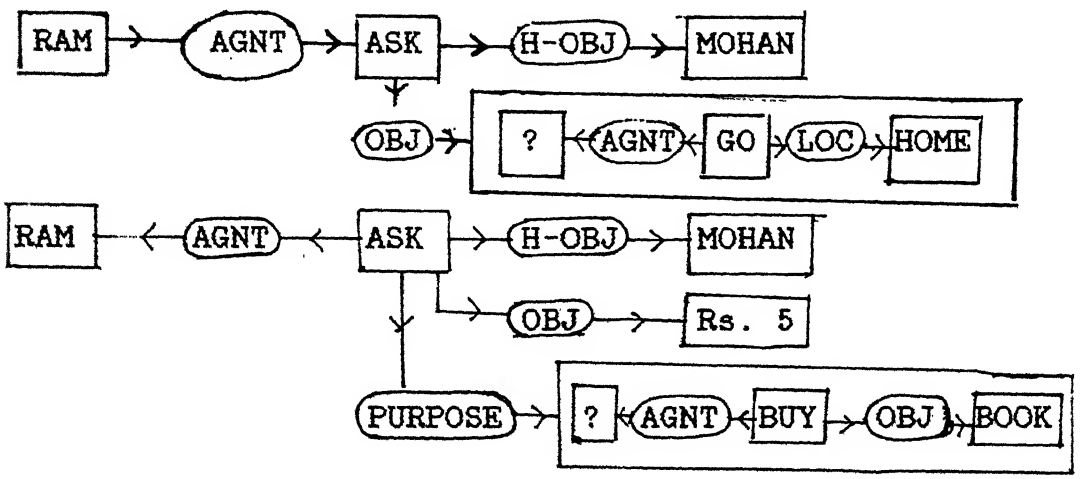
7.2 Example

Consider the following English sentences.

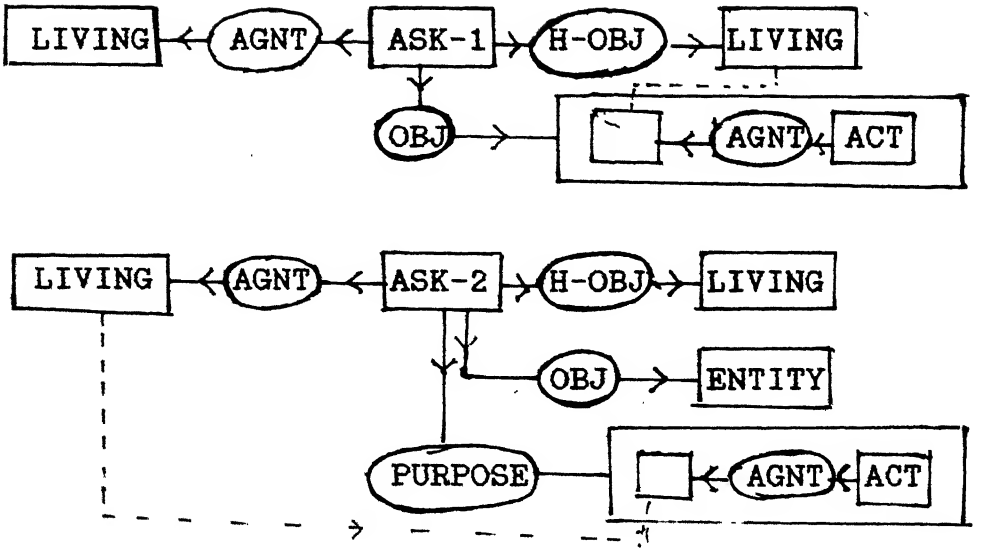
Ram asked Mohan to go home.

Ram asked Mohan Rs. 5 to buy a book.

In both the sentences a to-form is present. But the subject for the verb in the to-form is different for both the graphs. In the first sentence subject of going is, which is in to-form, Mohan, while in the second subject of buying is Ram. The parser generates the following graphs for the two sentences.



The question marks in the graphs have to be filled using the canonical derivation. The derivation is done by having the following graphs in the canonical basis.



Here graph 1 is derived from graph 3. Similarly graph 2 is specialization of graph 4. The question marks can be filled by making corresponding connections as in graphs 3 and 4.

The advantage of representing this knowledge in graph-form is the type-hierarchy. The ambiguity Resolution would be same for sentences containing the verb request (in place of ask in graph 1) or its subtypes.

Chapter 8

CONCLUSIONS

8.1 Summary of the thesis

The thesis describes implementation of the conceptual graphs. The canonical derivation using conceptual graphs is implemented. The canonical derivation is used to resolve ambiguities in natural language processing. The implementation also has a plausible truth derivation. It also has facilities for type abstraction and to use these abstractions to expand and contract the graphs.

The canonical derivation and plausible truth derivation are implemented using the following model. The model consists of a set of graphs which are static, Canonical basis for canonical derivation and schemas-set for plausible truth derivation. The set of graphs are used along with the type hierarchy and formation rules to derive the graphs.

8.2 Limitations and future work

The implementation does not cover logical inference. This is because the model described as above is not sufficient for logical inference.

(1) John has a book

(2) John gave the book to Mohan

Now suppose that these sentences are added to a set of facts. When we add the second sentence, the first fact becomes invalid (i.e. John does not not have the book any more). A new fact representing the fact that Mohan has the book has to be added to the fact set. In a more complex case, more changes have to be done to the graphs.

The model described in the first section has a static set of graphs which can be used for derivations in a mutually independent manner. This scheme cannot be extended to logical inference until time and modalities are handled correctly.

8.2.1 An alternate way to implement logical inference

This is to have two different kinds of knowledge as follows.

(1) A static set of assertions as in canonical derivation which consists of effects of implications In the example above it is having an assertion ' If John gave the book to Mohan , then John does not have the book and Mohan has the book '.

(2) A model of the world which represents an instance of the world that is being represented. As new knowledge comes into the system this model is modified according to implications of (1).

REFERENCES

[Sowa 84]

John F. Sowa , "Conceptual Structures ", Addison-Wesley ,1984.

[Charniak 85]

Eugene Charniak, Drew McDermott , "Introduction to Artificial Intelligence ", Addison-Wesley ,1985.

[Sowa 86]

John F. Sowa, Eileen C. Way , "Implementing a semantic interpreter using conceptual graphs " , pp 57-69 , IBM Journal for Research and Developmet , Jan 1986.

[Jean 86]

Jean Fargues, Marie-Claude Landau, Anne Dugourd, Laurent Catach "Conceptual graphs for semantics and knowledge processing ", pp 7Ø-79 , IBM Journal for Research and Development, Jan 1986.